Efficient Data Processing for large-scale Cloud Services

Marcel Tilly
European Microsoft Innovation Center
Aachen, Germany
Email: marcel.tilly@microsoft.com

Stephan Reiff-Marganiec
University of Leicester
Leicester, UK
Email: srm13@le.ac.uk

Helge Janicke
De Montfort University
Leicester, UK
Email: heljanic@dmu.ac.uk

***Abstract*—The cloud concept and its implementations are gaining in importance for systems that connect evermore new devices which in turn require communication with each other. In scenarios where we can find large numbers of data providers on one side and data consumers on the other side, such as in the Internet of Things, large scale sensor networks, machine to machine communication or even in social media, one emerging requirement is to process, procure, and provide information efficiently and with almost zero latency. This work is introducing new concepts to describe the flow of data to and from sources to cloud services in a formal way by limiting information flow with filtering concepts and combining data processing techniques adopted from complex event processing.**

***Keywords*-data filtering; policies; near-zero latency;**

I. Introduction

The cloud is an emerging technology that has significant growth and popularity over the last few years. Nowadays, the number of available services is increasing dramatically and the border between data and services is vanishing. Besides the classical static web pages there are also services and other data sources, such as sensors or phones. The term "Internet of Things" was coined, meaning that more and more data sources and services will be available in the future to provide a variety and broad set of new information, such as environmental information, geo location or social interactions. The cloud offers capabilities to support large-scale systems in terms of collecting data from all these various data sources and enabling the processing of that data. Data sources can be everything, such as smart phones, tablet PCs, sensors, cars etc. The cloud does not limit us to any specific type of data source. Thus, the cloud helps to provide a new set of services where we have on one side data sources and on the other side data sinks and in between a service in the cloud which is able to process these data in a very efficient way. Several aspects need to be discussed, analyzed and solved:

1) **Data provided by all data sources**: How to formally describe data and frequency of data send to the cloud service and to ensure balance between data accuracy and bandwidth.
2) **Real-time reasoning over data**: How to enable the cloud service to enable reasoning over data with almost zero-latency? This question is related to the issue about how fast and with which technology can the cloud service process data.

The interaction of data source (provider) and the cloud service is traditionally a push-model since the providers are sending actively data to the cloud. This is the first step towards faster processing of data in terms of providing results with low-latency. If the services are continuously pushing data to a cloud service there is a vast amount of overhead by unnecessarily transferring data – a waste of bandwidth. The cloud service knows about when it needs updated data and the provider knows about their context. Thus, the cloud service informs the provider under which changing situation (when) the providers should inform the data provider about the change of properties (what). What and when can be expressed with policy obligations which are injected into the providers. Thus, each provider will be responsible to make the projection from its own fine-grained, raw data to some more high-level, complex data the selector is interested in.

In this paper we provide an approach to overcome the problem of procurement, processing and provision of information in real-time in combination with optimizing the data traffic. We consider the use of complex event processing to enable a real-time view of service properties a fast and accurate view of their values in the cloud. Initial ideas from [19] are refined in this paper, and we add a formal syntax definition as a projection of the policy obligation injected on the providers. The novel contributions of the paper are (1) an architecture, data model and selection process to put the above into practice, and (2) a clear and well founded definition of policy obligations expressed at an abstract XML level as well as formally through ITL. Our approach can be realized as a cloud service and integrated with existing data sources. Section II presents a motivating example, enforcing the need for the mechanisms presented, while section III provides some essential background work. Section IV represents the core of the paper where we introduce the architecture and a formal description of the policy-based filtering approach based on temporal projection. Section VII points to some related work while section VIII concludes the paper and provides an outlook to further work.

II. Motivating Example

We introduce an example of social network management system to motivate our approach. The approach is not limited to this scenario and can be applied in a wide variety of applications where cloud services are matching a large set of potential consumer to providers, such as sensor network or logistics [19] , e.g.

1) Traffic sensor networks to monitor vehicle traffic on highways or in congested parts of a city.
2) Parking lot sensor networks to determine which spots are occupied and which are free.
3) Geo tracking of vehicles to support optimized routing of deliverables.
4) In fleet management, like taxi companies, to find a free taxi which is close by

In our example we are connecting virtual friends in real-life. Users of a social networks can provide their location information via a smart phone or tablet PC. Correlating this information with social network data, such as friends and their position, a service can send a notification back to the phone whenever one of the friends is close-by so that the user can meet his friend in real life. To do so, the user has to provide is geo position and information about his social network, his friends, to a service. In this scenario we have two types of data: (1) location data which is time dependent and might change frequently and (2) social data which can be considered static. The location data needs to be sent regularly to the cloud and might cause a lot of data traffic if we consider a large number of users. Assume there are Bob, Alice, Chris and Dave and they are all part of the same social network (see Figure 1). Alice is a friend of Dave, Chris and Bob, and Dave and Bob are also friends. While Bob is sitting at home, Alice is traveling with the metro from A to B, Dave is going by car from C to D, and Chris is in a rush and drives from E to F. Since Bob's geo position is almost static he only needs to provide his position once. In contrast, Alice, Chris and Dave are moving and hence need to provide frequent updates of their geo positions. The cloud service itself collects and processes all data. In case there is a match the service can inform the users. So that for example Alice and Dave can be informed at time t and position X that they are close to each other so that they can take this chance to talk to each other face to face. Since Chris is in a hurry, he has no time to meet and talk to one of his friends, he has set his availability to false. Although he could meet Alice as well in Y, his device should not provide any position data since there is no time anyway. The service would not propose that Chris and Dave should meet in Z because they are not friends.

This scenario shows (1) how different kind of properties (here: availability and geo location), (2) properties of different services (here: geo position and social network) are used to match user data, and (3) that users have to pro-

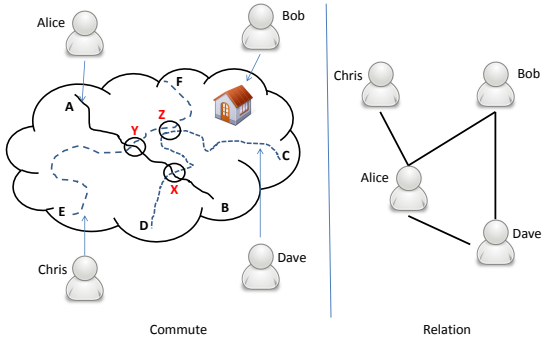

Figure 1. Friend-Near-by Sample

actively inform the service about their location to enable fast and reliable responses to customer requests. Thus, it makes sense to have a rule running on the device which defines which data should be used (*event*), when data should be sent (*condition*) and what data should be send (*action*). While each specific rule is very specific to a scenario, they help to balance between optimizing data traffic and data accuracy in a cloud system (and actually in terms of privacy it is even better to keep less).

Furthermore, the geo location is data which changes rapidly and it does not make sense to store all data because it is only short-lived and hence only the current values are relevant when a service has to be selected.

III. Background

This section introduces the basic ideas and formal descriptions which we combine to improve processing of data in real-time. As we formally model the temporal abstraction, we will also provide a short introduction to interval temporal logic.

In this work the policies are modeled using the well-understood Event-Condition-Action paradigm [8], [21], [22]. The novelty of the policies is that they use temporal conditions that describe the distance between two consecutive actions that push data to aggregating services and to correlate different data streams in the cloud by using the same paradigm. Informally a policy is a set of rules of the following structure:

```
<Policy> <!-- send to Service -->
  <Rule>
    <Target>...</Target>
    <Event>...</Event>
    <Condition>...</Condition>
    <Action>...</Action>
  </Rule>
  <Rule> ... </Rule>
</Policy>
```

Expressions	
$e ::=$	$\mu \mid a \mid A \mid g(e_1, \ldots, e_n) \mid \bigcirc v \mid \mathsf{fin}\ v$
Formulae	
$f ::=$	$p(e_1, \ldots, e_n) \mid \neg f \mid f_1 \wedge f_2 \mid \forall v \bullet f \mid \mathsf{skip} \mid f_1 ; f_2 \mid f_1 \Delta f_2$

Figure 2. Syntax of ITL

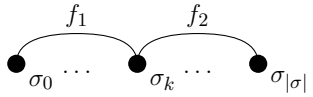

Figure 3. Informal Semantics of $f_1 ; f_2$

The <Target> of a rule is a list of services on which the <Action> of the rule is invoked if the rule is triggered. The <Event> of a rule is a event descriptor that determines when the <Condition> of the rule is evaluated. The descriptor is a predicate build from primitive events (e.g. a GPS-Update) that are domain dependent and defined in the service description. Conceptually the event descriptor describes an abstraction of the event trace over which the <Condition> is evaluated. The <Condition> describes the distance between events that are communicated downstream to aggregating services as a temporal formula. The syntax that is used is an XML representation of Interval Temporal Logic formulae that is described in III-A

A. ITL

The key notion of ITL [3] is an *interval*. An interval σ is considered to be a (in)finite sequence of states $\sigma_0, \sigma_1 \ldots$, where a state σ_i is a mapping from the set of variables Var to the set of values Val. The length $|\sigma|$ of an interval $\sigma_0 \ldots \sigma_n$ is equal to n (one less than the number of states in the interval, so a one state interval has length 0).

The syntax of ITL is defined in Figure 2 where μ is a constant value, a is a static variable (does not change within an interval), A is a state variable (can change within an interval), v a static or state variable, g is a function symbol and p is a predicate symbol. The syntax is based on [3], however uses the projection operator $f_1 \Delta f_2$ as primitive and derives the operator f^* as introduced in [14].

The informal semantics of the most interesting constructs are as follows:

- skip: unit interval (length 1, i.e., an interval of two states).
- $f_1 ; f_2$: ("chop") holds if the interval can be decomposed ("chopped") into a prefix and suffix interval, such that f_1 holds over the prefix and f_2 over the suffix, or if the interval is infinite and f_1 holds for that interval. Note the last state of the interval over which f_1 holds is shared with the interval over which f_2 holds. This is illustrated in Figure 3.
- $f_1 \Delta f_2$: ("projection") is defined to be true on an interval σ iff two conditions are met. First, the formula f_2 must be true on some interval σ' obtained by projecting some states from σ. Second, the formula f_1 must be true on each of the subintervals of σ bridging the gaps between the projected states.
  An example is depicted in Figure 4.

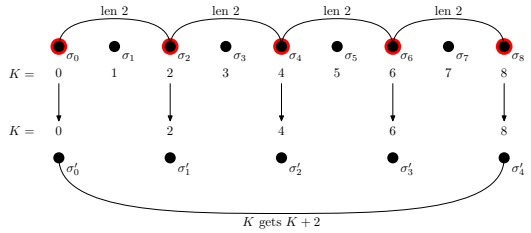

Figure 4. Example of Temporal Projection

  In the interval σ the value of K increases from 0 to 8 in steps of one. The interval σ satisfies $(\mathsf{len}(2))\Delta(K\ \mathsf{gets}\ K+2)$. $(\mathsf{len}(2))$ is true if the interval is of length two and $(K\ \mathsf{gets}\ K+2)$ is true if the K increases by 2 from state to state. The gaps between the projected states (highlighted in red) are bridged by the formula $\mathsf{len}(2)$. The formal definition of this operator is given in [14].
- $\bigcirc v$: value of v in the next state when evaluated on an interval of length at least one, otherwise an arbitrary value.
- $\mathsf{fin}\ v$: value of v in the final state when evaluated on a finite interval, otherwise an arbitrary value.

1) Derived Constructs.: The following lists some of the derived constructs used in the remainder of this paper. The binary operators $\vee$ (or) and $\supset$ (implication) are derived as usual.

$\bigcirc f \mathrel{\widehat{=}} \mathsf{skip} ; f$ (read "next f"), means that f holds from the next state. Example: $\bigcirc(X = 1)$: Any interval such that the value of X in the second state is 1 and the length of that interval is at least 1.

$\mathsf{more} \mathrel{\widehat{=}} \bigcirc\mathsf{true}$ means the non-empty interval, i.e., any interval of length at least one.

$\mathsf{empty} \mathrel{\widehat{=}} \neg\,\mathsf{more}$ means the empty interval, i.e., any interval of length zero (just one state).

$\mathsf{inf} \mathrel{\widehat{=}} \mathsf{true} ; \mathsf{false}$ means the infinite interval, i.e., any interval of infinite length.

$\mathsf{finite} \mathrel{\widehat{=}} \neg\,\mathsf{inf}$ means the finite interval, i.e., any interval of finite length.

$\Diamond f \mathrel{\widehat{=}} \mathsf{finite} ; f$ (read "sometimes f"), i.e., any interval such that f holds over a suffix of that interval. Example: $\Diamond X \neq 1$: Any interval such that there exists a state in which X is not equal to 1.

$\Box f \mathrel{\widehat{=}} \neg \Diamond \neg f$ (read "always f"), i.e., any interval such

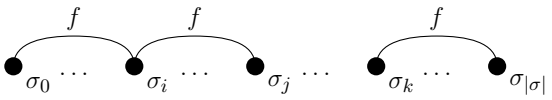

Figure 5. Informal Semantics of f^*

that f holds for all suffixes of that interval. Example: $\Box(X = 1)$: Any interval such that the value of X is equal to 1 in all states of that interval.

$\mathsf{fin}\ f \mathrel{\widehat{=}} \Box(\mathsf{empty} \supset f)$ defines the final state, i.e., any interval such that f holds in the final state of that interval.

$\mathsf{halt}\ f \mathrel{\widehat{=}} \Box(\mathsf{empty} \equiv f)$ terminate the interval when f holds.

$\exists v \bullet f \mathrel{\widehat{=}} \neg \forall v \bullet \neg f$ existential quantification.

$$\mathsf{len}(e) \mathrel{\widehat{=}} \begin{cases} \mathsf{false} & \textbf{if } e < 0 \\ \mathsf{empty} & \textbf{if } e = 0 \\ \mathsf{skip}\,;\mathsf{len}(e-1) & \textbf{if } e > 0 \end{cases}$$ holds if the interval length is e.

$v\ \mathsf{gets}\ e \mathrel{\widehat{=}} \Box(\mathsf{more} \supset (\bigcirc v) = e)$ gets, i.e., in every state except the initial state the variable v will be assigned the value of e evaluated in the previous state.

$f^* \mathrel{\widehat{=}} f \Delta \mathsf{true}$ (read "f chopstar") holds if the interval is decomposable into a finite number of intervals such that for each of them f holds, or the interval is infinite and can be decomposed into an infinite number of finite intervals for which f holds. This is illustrated in Figure 5.

IV. Architecture

It is quite challenging to obtain an accurate view of data coming from a huge number of different sources (here: providers). We can define a need for an concept delivering responses with low latency based on dynamic service properties at any time to users from huge lists of providers.

Basically, we propose to extend the publisher and subscribe pattern with an mediator service in the cloud which consumes data (in form of events) and produces rich results to the subscriber (Figure 6). By running the mediation service in the cloud the approaches benefits automatically from elastic scaling capabilities of the cloud infrastructure.

Our approach can be adopted easily as it only requires the addition of two interfaces:

1) The publisher endpoint is exposed on the service side to which the consumer can register or subscribe to events
2) the subscriber endpoint is exposed by the *cloud service* to enable the services to fire events in a fire and forget fashion (see Figure 6).

The publisher interface which enables the registry to subscribe to a set of dynamic properties provides two

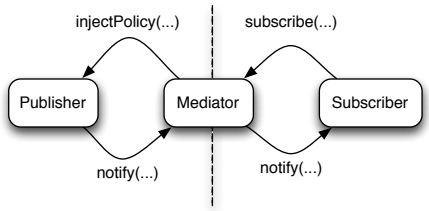

Figure 6. Mediation Pattern

operations:

$$injectPolicy(policyObligation) : Id$$

1) policyObligation: the policy obligation describes the topic to be subscribed to, the refresh interval, and the state changes which trigger event notification.
2) Id: Unique registration id for the subscription

$$unsubscribe(Id)$$

1) Id: Unique registration id

The mediator interfaces enables subscriber to get specific notifications.This is following the classical pub-sub pattern:

$$subscribe(topic) : Id$$

and

$$unsubscribe(Id)$$

The mediator and the subscriber are able to receive events. They offer the following interface:

$$notify(Event)$$

An *Event* is a tuple of values $event = \langle se, ts, te, p \rangle$, containing the service endpoint address se, time information ts and te, and the payload p. The time information defines the valid start time ts and end time te of the event and the payload is defined by the type of the subscribed topic. For example the *GeoLocation* could be defined as record with *Longitude* and *Latitude*, both of the XML schema type *xs:int*.

As described in [15] processing of streaming data is an important practical problem that arises in time-sensitive applications where the data must be analyzed as soon as they arrive, or where the large volume of incoming data makes storing all data for future analysis impossible.

As a central instance we use a *Mediator* (see Figure 7) running in the cloud. This *Mediator* encapsulates the processing of the incoming request from the subscriber side and the incoming events from the provider side. The *Mediator* is a service and exposed operations (methods) map internally to specific queries. Thus, during runtime the *Mediator* is receiving continuous streams of events from providers. Then, in cases there is a mapping subscribers are notified.

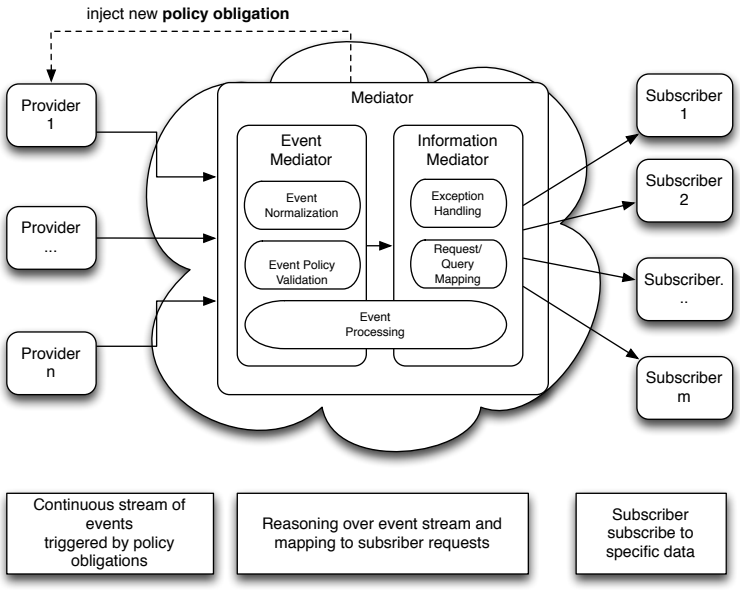

Figure 7. Mediation service architecture

An event will contain metadata and payload. The metadata contains information about the time when the event was created on the publisher side. The schema of the subscribed topic, such as geo location, temperature or vibration, defines the payload. New policy obligations are injected via the *Mediator* into the correct service (publisher).

A. Event Mediator

The *Event Mediator* exposes an endpoint to collect all incoming events from registered providers. Its responsibility is to normalize the incoming data streams.

B. Information Mediator

The *Information Mediator* maps subscriber request to queries on continuous event streams provided by the *Request Mediator*. On the subscriber side the framework still offers a normal Web Service interface, which internally needs to be transformed into a query, which is executed over the event stream. The *Information Mediator* also ensures the quality of the events from event streams, such as duplicated events or out-of-order events. Here, our approach benefits from the existing work on complex event processing (CEP), such as [12] or [13].

C. Filtering at the source

To control the event flow from providers to the mediator the services are accepting policy obligations as filtering rules. These obligations are defining which state changes within a service (on the source) trigger an event (such as "temperature $> 50.2C$") and the expected interval (refresh). The expected interval would then also be used within an event so that the start time is set when the event is issued on the service and the end time is defined by the refresh interval. Section V will deal with the specification of the required filters in detail. Being able to set the event interval rate and condition helps to fine-tune the system to obtain the appropriate balance between data accuracy, response time and data traffic.

V. Event Policies

We introduce event policies by a collection of representative examples:

Example 1 (Event Filter based on Explicit Time)**.**

```
<Policy> <!-- send to Service -->
  <Rule>
    <Target>Aggregation Service</Target>
    <Event>GPSUpdate()</Event>
    <Condition>
      <NEXT>
      <HALT>
        <GT>
          <FIN>
            <VAR Id="Time"/>
          </FIN>
          <SUM>
            <VAR Id="Time">
            <CONST type="xs:int">1000</CONST>
          </SUM>
        </GT>
      </HALT>
      </NEXT>
    </Condition>
    <Action name="notify">
      <GPSLocation id="x"/>
    </Action>
  </Rule>
</Policy>
```

This policy would be stipulated by the *Aggregation Service* (Mediator) to filter the *GEOLocation* updates send by the "Publisher" *Service*. The single obligation rule is only evaluated on *GPSUpdate()* events local to the *Service*.

The condition states that the time between *notify* Actions is at least *1000ms* and that on the first occurrence of a *GPSUpdate()* thereafter the *GEOLocation* is pushed to the target, in this example the *Aggregation Service*.

Another example would lead to an update being send to the aggregation service whenever the Euclidean distance between the last update and the current position exceeds 50m:

Example 2 (Update based on Dynamic Attributes)**.**

```
<Policy> <!-- send to Service -->
  <Rule>
    <Target>Aggregation Service</Target>
    <Event>GPSUpdate()</Event>
    <Condition>
      <NEXT>
      <HALT>
        <GT>
          <FUNCTION name="EuclideanDistance">
            <GPSLocation id="x"/>
            <FIN> <GPSLocation id="x"/> </FIN>
          </FUNCTION>
          <CONST type="xs:int">50</CONST>
        </GT>
      </HALT>
      </NEXT>
    </Condition>
    <Action name="notify">
      <GPSLocation id="x"/>
    </Action>
  </Rule>
</Policy>
```

Building on earlier work [11] in the context of access control, the condition of a policy is a temporal description of the filter that is applied to the selected event stream.

Conceptually the service is filtering its event stream as depicted in Figure 8. Here the *Service* is processing its own stream of events (Service Events) that define its internal behaviour. The *Event* trigger in the definition of rules selects a sub-stream that contains only those states in the Service's behaviour at which the *Event* was raised. The condition in the rule is evaluated over this filtered event-stream and further restricts the behaviour based on the condition expressed in the rules. The condition is an interval temporal logic formula that defines the distance between any two selected states. In theses states an action is performed (e.g. "notify") that exposes information to other connected services, in this case the *Aggregation Service*. As a result, the policy determines the externally observable behaviour of the service. Connected services can influence this behaviour by updating the event policies via the `injectPolicy()` operation.

Services that support these policy filters can be combined into hierarchies as shown in Figure 8, yielding service compositions that fuse and filter information defined in their NFP schemas based on policies. The information will be provided on a PUSH model with policies determining the frequency and conditions of updates, yielding a flexible, policy-based publish-subscribe infrastructure.

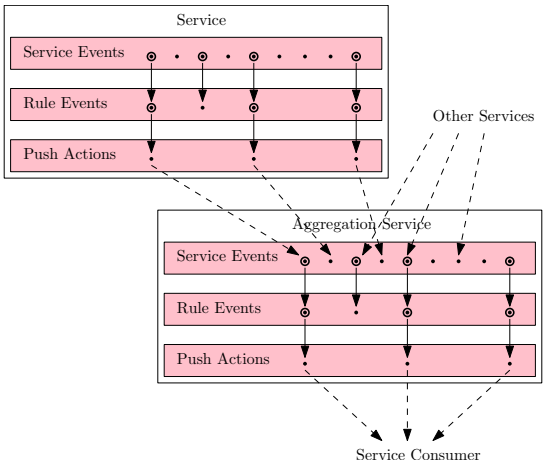

Figure 8. Policy-Based Event Stream Filter

A. Formal Model

Let each service $s \in Services$ be defined over a continuous stream σ_s of events $e_i \in Events_s$, observed by the service s. This is modelled by representing σ_s as an ITL interval and $Event_s$ as a set of propositional state variables that indicate the occurrence of events (recall that state variables can change their value from state to state). This model allows for the concurrent occurrence of events, e.g. $e_i \wedge e_j$ $(i \neq j)$, and only captures the sequence of events, rather than their absolute timing. The creation time of the event is stored explicitly as part of the event tuple and can be referred to in the conditions of policy rules.

An event is described as a tuple $\langle se, ts, te, p \rangle$, denoting the service se creating the event, the time-stamp when the event was created ts, te (based on the clock of s) and an optional payload p. We use the notation $e.se, e.ts, e.te$ and $e.p$ when referring to a specific element of an event tuple e.

B. Event Policy Validation

Evaluating the policy pol_s of the service s against this interval is a two stage process.

1) Stage 1: First, for every rule $r \in pol_s$ an abstraction of the interval σ_s is generated based on the *Event* trigger evt_r of the rule r. Currently we only consider single event triggers, however the formal model is supporting combined events such as $e_i \wedge e_j$ or state formulae (i.e. ITL formulae that do not contain temporal operators). Conceptually this stage is generating an abstracted interval $\sigma_{s,r}$ of the interval σ_s that contains only those states in which evt_r is true. This is depicted in Figure 9.

2) Stage 2: Second, for every rule r the condition of the rule cnd_r is evaluated against the corresponding abstracted interval $\sigma_{s,r}$. The condition defines the distance between two consecutive actions triggered by the same rule. This means that the temporal formula cnd_r must hold over the

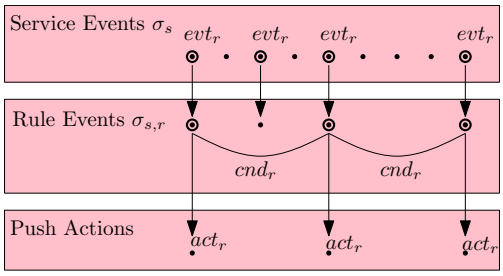

Figure 9. Policy Rule Evaluation

subintervals of $\sigma_{s,r}$ bridging the gaps between the projected states. This is depicted in Figure 4.

Formally this means that the policies relate the service's event trace, viz. the interval σ_s to actions that are performed by the service as follows:

$$\sigma_s \models \bigcirc\mathsf{halt}\,(evt_r)\Delta(cnd_r\Delta\Box act_r)$$

Here $\bigcirc\mathsf{halt}\,(evt_r)\Delta f$ conceptually yields the abstracted interval $\sigma_{s,r}$ over which the policy rule is evaluated. The condition cnd_r of the rule then bridges between two consecutive actions that are performed as a consequence of the rule.

The rationale for separating the two steps is that the filtering of event streams based on simple events (evt_r) can be implemented very efficiently, whereas the complexity of the evaluation of the conditions cnd_r is more complex and can in certain cases grow linearly with the number of states that are bridged. Thus the initial reduction using the event filter reduces the complexity of the latter evaluation.

The overall service specification is then constructed from this as:

$$\sigma_s \models \bigwedge_{r\in pol_s} \bigcirc\mathsf{halt}\,(evt_r)\Delta(cnd_r\Delta\Box act_r)$$

The specification of act_r is not detailed here and we only consider that the relevant action is initiated in that state of the service interval.

The model can be implemented straightforwardly from its semantics using Tempura [9], [3], an executable subset of ITL. For example:

```
/* run */ define example() = {
exists Evts :
{ /* create test event trace for the service */
 list(Evts,3) and stable(struct(Evts)) and evtmodel(Evts) and
 { /* example rule evaluation */
   (next halt(Evts[0]=1)) /* selecting events Evts[0] */
   proj{ /* show selected events, testing only */
     always format("Evts[0] = 1\n") and {
     len(2) /* select every second event only */
     proj{  /* show selected events, testing only */
       always format("Action on every 2nd Evts[0].\n")
}}}}}}.

set assign_ahead = false.
define evtmodel(Evts) = {
 Evts = [1,1,0] and skip ; Evts = [0,0,1] and skip ;
 Evts = [1,1,0] and skip ; Evts = [1,0,1] and skip ;
 Evts = [0,0,0] and skip ; Evts = [0,0,1] and skip ;
 Evts = [1,0,0] and skip ; Evts = [1,0,0] and empty
}.
```

Here three events are modelled for the service, and an example trace is generated by the function `evtmodel(Evts)`. AnaTempura can be run in a run-time verification mode and could receive these events from an external program. The event trigger for the encoded rule is `Evts[0]`, where a value of `1` indicates that the event occurred. This is encoded in the first projection condition `(next halt(Evts[0]=1))`, which in effect generates the more abstract interval $\sigma_{s,r}$ over which the second projection is taking place. In this example the temporal condition is selecting every second of the events (`len(2)`) on which the action of the rule is triggered. In this proof of concept only a statement is printed out to the screen, but instead a message could be easily send to another service. The above code can be readily executed in AnaTempura (available at http://www.cse.dmu.ac.uk/STRL/ITL/)

VI. Example (cont'd)

Let us go back to the motivating scenario (see section II) to illustrate the presented theory. In this example we have the users who want to get a notification when one of their friends is near by. To get this notification the user has to subscribe to the mediator. Technically the *Mediator* is responsible to handle all subscriptions from users and map it to the geo location event streams coming from all user devices. The ECA rule running on the mediator is receiving two data streams: (1) is the stream of the friend relationship and (2) the stream of users' geo locations. Since, the friend relationship stream is considered static the events in this stream are having infinite live time. The geo location stream contains different events for each user and the live time of each event is defined by the policy injected in each device, such as the speed on changing the geo position.

Each node (user's phone) is sending an event to the *Mediator* whenever it moves more than 50 meters (see Example 2).

The ECA policy running on the mediator is collecting these notifications and correlates this geo location data with

the friend relation information so that we can notify each user if one of his friends is close by, e.g.

- we want to notify Alice when one of Alice's friends is closer than 50 meters

This can be translated into the following an ECA policy:

```
Event: GeoLocationUpdate(x)
Condition: friend(Alice, X) and distance(alice,X)
Action: Notify Alice of X
```

In our example we do have Alice, Bo, Chris, and Dave (see Figure 10). At time Y Chris is at position Y but Alice is not, so there is no action triggered. At time Z Chris and Dave are at the same position but they are not friends. Again no event is issued. At time Z Alice and Dave are at position X and they are friends so that both are getting a notification.

VII. Related Work

While there is work complex even processing and temporal logic there is no work as far as we are aware which combines these approaches to enable fast data processing for large scale systems in the cloud. Bonifati et al. [2] describes a very interesting approach for using active rules for pushing reactive services. But it does not take into account temporal aspects or states. Roitman et al. [16] presents a framework for satisfaction of complex data needs involving volatile data. But the focus is on pull-based environments. We believe that our approach is more promising for large scale systems and the cloud. With push based systems, data is pushed to the system and the research focus is mainly on aspects of efficient data processing, where load shedding techniques [20] can be applied in order to control what portions of the pushed data to process and to increase latency. Such systems include publish-subscribe (pub/sub) ([5]), stream processing ([1]), and complex event processing, however there is no consideration of bandwidth consumption.

The use of Event-Condition-Action (ECA) rules is well established in Data-Stream Processing applications [8], [4] and ECA-based policy languages [21], [22] are used to govern the behaviour of systems on the basis of these rules to control and manage distributed systems [17]. In our work, we are however mainly concerned about the selection and propagation of events in a P2P infrastructure. The formalisation of event policies in this work differs from traditional ECA rules in that the condition does not only describe a Boolean combination of events, but can address the history of a selected event stream that allow to specify the *distance* between propagated events using ITL [3]. Data Stream processors such as SNOOP [4] and successors already use event histories for detecting the order of events, making this a natural model for expressing policies that also allows for the efficient enforcement of such policies [10]. The semantics of event-policies is based on temporal projection [14] as this is a natural abstraction technique for complex system specifications. Other work by Duan et.al. [6], [18] on propositional projection temporal logic would provide alternative formalisations of projection, but lead to a more complex formalisation of the event policies without apparent gain in this application context. The use of policies together with a mediator has also been suggested in a different context by Edge et.al. [7] where they focus on the mining institutional transaction data for fraudulent activities. However, their use of policies is targeted to this particular application domain and is focused on the detection of events, whereas we address the problem of event filtering and propagation as part of an infrastructure for cloud based systems.

VIII. Conclusion and Future Work

We presented a new approach which combines event processing based on interval temporal logic for service selection approaches to enable a fast data processing in the cloud. Our approach investigates data processing and accuracy for a huge number of potential providers and highly dynamic service properties. By combining ECA with ITL we have grounded dynamic service properties on a valid formal model. We presented a way to use ITL to express policy obligations as ECA rules which should be executed close to the sources to enable an accurate view of a large scale system at any point in time and to reply to subscribers with almost zero latency. The sources (in our example geo location of user) are notifying the mediation service about any state change defined by policies thus the mediation service can (1) reason about the incoming streams and reply immediately to consumer requests and (2) the mediation service can make assumptions in terms of missing data and forecast likely future behaviour. In contrast to our previous work we are using in this work the ECA policy not only on the nodes as injected policies but also on the mediator to process streams of data and to correlate it for fast data processing.

References

[1] D. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, A. Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Xing, R. Yan, and S. Zdonik, *Aurora: a data stream management system*, Proceedings of the 2003 ACM SIGMOD international conference on Management of data (New York, NY, USA), SIGMOD '03, ACM, 2003, pp. 666–666.

[2] Angela Bonifati, Stefano Ceri, and Stefano Paraboschi, *Pushing reactive services to XML repositories using active rules*, Computer Networks **39** (2002), 645–660.

[3] A. Cau, B. Moszkowski, and H. Zedan, *The ITL homepage: http://www.cse.dmu.ac.uk/STRL/ITL*, Tech. report, Software Technology Research Laboratory, De Montfort University, 2011.

[4] S. Chakravarthy and D. Mishra, *Snoop: an expressive event specification language for active databases*, Data Knowl. Eng. **14** (1994), 1–26.

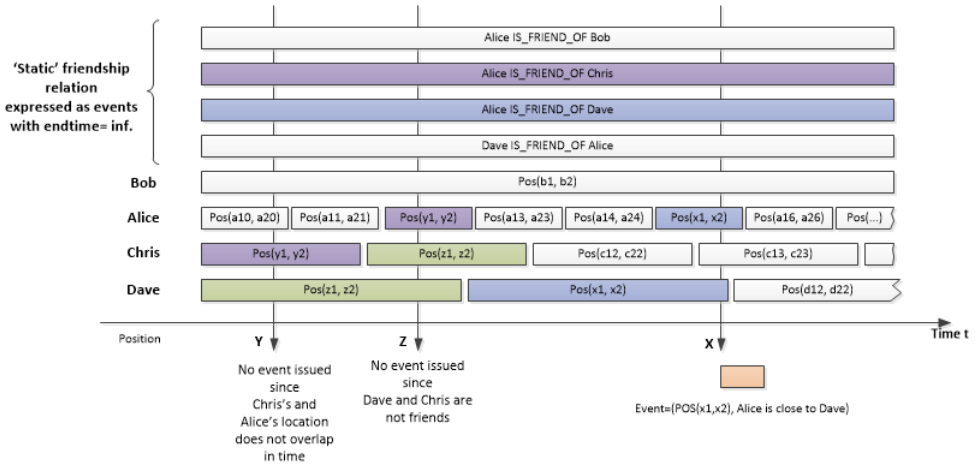

Figure 10. Event Stream Reasoning

[5] A. Demers, J Gehrke, M Hong, M Riedewald, and W. White, *Towards expressive publish/subscribe systems*, Advances in Database Technology-EDBT 2006 (2006), 627–644.

[6] Zhen-Hua Duan and Maciej Koutny, *A framed temporal logic programming language*, J. Comput. Sci. Technol. **19** (2004), 341–351.

[7] Michael Edge, Pedro Sampaio, Oliver Philpott, and Mohammed Choudhary, *A policy distribution service for proactive fraud management over financial data streams*, Services Computing, IEEE International Conference on **2** (2008), 31–38.

[8] Charles L. Forgy, *Expert systems*, Expert systems (Peter G. Raeth, ed.), IEEE Computer Society Press, Los Alamitos, CA, USA, 1990, pp. 324–341.

[9] Roger William Stephen Hale, *Programming in Temporal Logic*, Ph.D. thesis, Trinity College, University of Cambridge, October 1988.

[10] Helge Janicke, Antonio Cau, François Siewe, and Hussein Zedan, *Deriving Enforcement Mechanisms from Policies*, Proceedings of the 8th IEEE international Workshop on Policies for Distributed Systems (POLICY2007), June 2007, pp. 161–170.

[11] Helge Janicke, Antonio Cau, François Siewe, Hussein Zedan, and Kevin Jones, *A Compositional Event & Time-based Policy Model*, Proceedings of POLICY2006, London, Ontario, Canada (London, Ontario Canada), IEEE Computer Society, June 2006, pp. 173–182.

[12] Ling Liu, Calton Pu, and Wei Tang, *Continual queries for internet scale event-driven information delivery*, Knowledge and Data Engineering, IEEE Transactions on **11** (2002), no. 4, 610–628.

[13] David Luckham, *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*, Addison-Wesley Longman, Amsterdam, 2002.

[14] Ben Moszkowski, *Compositional Reasoning about Projected and Infinite Time*, Proceedings of the 1st IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'95) (Fort Lauderdale, Florida), IEEE Computer Society Press, Novenmber 1995, pp. 238–245.

[15] A Riabov and Z Liu, *Scalable planning for distributed stream processing systems*, Proceedings of ICAPS, vol. 06, 2006.

[16] Haggai Roitman, Avigdor Gal, and Louiqa Raschid, *Web Monitoring 2.0: Crossing Streams to Satisfy Complex Data Needs*, Proceedings of the 2009 IEEE International Conference on Data Engineering, IEEE Computer Society, 2009, pp. 1215–1218.

[17] M. Sloman, *Policy driven management for distributed systems*, Journal of Network and Systems Management **2** (1994), 333–360.

[18] Cong Tian and Zhenhua Duan, *Complexity of propositional projection temporal logic with star†*, Mathematical. Structures in Comp. Sci. **19** (2009), 73–100.

[19] Marcel Tilly and Stephan Reiff-Marganiec, *Matching customer requests to service offerings in real-time*, Proceedings of the 2011 ACM Symposium on Applied Computing, SAC '11, ACM, 2011, pp. 456–461.

[20] Y.C. Tu, S. Liu, S. Prabhakar, and B. Yao, *Load shedding in stream databases: a control-based approach*, Proceedings of the 32nd international conference on Very large data bases, VLDB Endowment, 2006, pp. 787–798.

[21] K. Twidle, E. Lupu, N. Dulay, and M. Sloman, *Ponder2 - a policy environment for autonomous pervasive systems*, Policies for Distributed Systems and Networks, 2008. POLICY 2008. IEEE Workshop on, june 2008, pp. 245 –246.

[22] A. Uszok, J. Bradshaw, R. Jeffers, N. Suri, P. Hayes, M. Breedy, L. Bunch, M. Johnson, S. Kulkarni, and J. Lott, *KAoS policy and domain services: toward a description-logic approach to policy representation, deconfliction, and enforcement*, Proceedings POLICY 2003 Policies for Distributed Systems and Networks, June 2003, pp. 93–96.